Developing Handle System Web Services at Cornell University.

Abstract

Recognizing the importance of consistently implementing persistent identifiers (PIDs), a group of librarians at Cornell University evaluated several PID strategies and chose to implement the Handle System. As a member of that group, the author immediately perceived cultural and technical challenges to adopting handles that included accommodating diverse computing platforms. The development of cross platform Web Services and corresponding client libraries are explored as a way to address these challenges and create new opportunities for future development and maintenance of digital collections.

A Persistent Identifier Primer

Existing methods for identifying resources are ephemeral because these methods typically facilitate access by locating those resources within their specific delivery mechanisms. Uniform Resource Identifiers (URIs) provide a familiar example of such identifiers that break as resources change location, as underlying delivery methods change, or as resources simply disappear.

Alternately, persistent identifiers (PIDs) strive to unambiguously identify resources for the purpose of citation and access. PIDs are therefore:

- assigned to single, distinct resources,
- independent of location and the underlying delivery technology for those resources,
- globally unique, and
- persistent, that is, each PID always identifies the same resource, even when that resource ceases to exist or is no longer accessible.

Cornell Chooses the Handle System

In the past, Cornell University Library (CUL) has implemented Persistent Uniform Resource Locators (PURLs) for its electronic resources, handles within DSpace, and even more often, no PID approach has been implemented in new systems. As a result, several librarians within CUL have advocated for a more consistently implemented PID strategy primarily to support the creation and preservation of digital collections. This advocacy culminated in the creation of a working group in the first quarter 2006 charged to recommend a PID strategy for CUL going forward, with immediate application to several digital repository and preservation systems then being developed. The author was one of the librarians that participated in this working group.

In addition to reviewing general identifier strategies and issues, as well as potential intersections between PIDs and other standards, such as OpenURLs, and the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH), the PID working group at CUL compared and contrasted PURLs, Archival Resource Keys (ARKs) and the Handle System among other approaches. The group ultimately recommended the Handle System based on factors that included its general maturity and installed based, its fit for CUL's projects, and the existing knowledge of this approach within CUL.

Soon after this preliminary recommendation was made, a local handle server instance was installed to further evaluate the technical and organizational requirements for implementing and maintaining handles as well as to determine how to integrate handles into CUL's digital collection development lifecycle. Not the least of the author's concerns during this process is the cultural impact of adopting handles as an architectural best practice within CUL, which may add significant new requirements to projects already on aggressive schedules. The author believes that an adoption strategy of advocacy and encouragement rather than one of stipulation and enforcement is more likely to be successful, so the author endeavored to make administering handles as painless as possible for the developers impacted by this decision. However, this goal is complicated by the fact that CUL maintains systems built on multiple architectural stacks and will surely continue to adopt new technologies into the future.

Developing Handle System Web Services

Once the initial CUL handle server installation was complete, the author began to integrate handles into a pilot preservation system using the Handle.net Software Client Library's (HCL) application programming interface (API) [1]. But the HCL API was implemented at a lower level than had been anticipated; power and flexibility in the API were purchased at the expense of ease of use for developers.

The HCL API also assumes the Handle System will be accessed in a Java based environment, or at least an environment that can integrate Java components. Although typically methods exist for one programming environment to communicate with another, these methods are often inefficient and architecturally fragile. Because CUL has legacy systems written in languages other than Java that may need to incorporate handles, and because the architectures of future systems cannot be predicted with certainty (including the Handle System itself), interacting with the Handle System using a standard protocol equally accessible to all environments, independent of the Handle System implementation, is ideal.

Fortunately, the HCL source code is available in the HCL distribution, complete with sample interfaces, and with this source code as documentation, the author harnessed the power of the HCL API to create more convenient interfaces to CUL's local handle server. The immediate goals were to hide the complexity of the Handle System's authentication infrastructure behind a platform independent interface that would expose create, read (or resolve), update and delete (CRUD) semantics for administering handles. Specifically, the following use cases are implemented, both for single and batched requests:

- A client supplies a URI and receives the corresponding handle (PID creation).
- A client supplies an existing handle and receives its corresponding URI (PID resolution).
- A client supplies an existing handle and a URI, and receives a confirmation of its update (PID modification).
- A client supplies an existing handle and receives confirmation of its removal (PID deletion).

(The PID creation service currently assigns institutionally unique identifiers via a Nice Opaque Identifier (NOID) minter [2]. This institutionally unique identifier along with the institutional prefix registered with the Global Handle System and the local name for the handle server installation, ensures that the generated handle values are globally unique.)

Using the bundled HCL command line interface (CLI) source code as a guide, the author developed these four use cases as Web Services that enable client software on any platform to communicate with the local handle server by sending and receiving standard XML messages. The CLI source devotes a significant block of complex Java code to manipulating passwords as well as public and private key pairs to authenticate handle server requests. So, the author first factored out the authentication code, both to minimize the redundancy of the API calls across the Web Services and also to abstract those API calls behind interfaces to reduce their complexity. This code eventually became the base class from which each of the classes implementing the CRUD services inherits.

The author employed standard Java tools for refactoring, that is, making changes to the design of the code without changing its underlying behavior, along with a Test Driven Development (TDD) approach, to transform the CLI source into four distinct CRUD classes, each re-using the base authentication code. The author then used the open source Apache Axis Web Services toolkit [3] to transform these otherwise normal classes into full fledged Web Services that use Simple Object Access Protocol (SOAP) for XML messaging. These Web Services could then be deployed inside of a standard Apache Tomcat servlet container [4] and made available on the Web.

Finally, the author had to decide how to authenticate the Web Services themselves. The Handle System authentication was now hidden behind the Web Services, and while these authentication mechanisms still police the interaction between the Web Services and the local CUL handle server, without further development, the Web Services themselves would be exposed to anonymous access. For expediency and ease of use, the author implemented a simple Internet Protocol (IP) address based authentication mechanism to allow code running on trusted machines to access the Web Services. For each Web Service request that results in a change on the local handle server (encompassing all requests except for PID resolution), the IP address of the incoming request is matched against a file listing valid IP addresses. The originating IP address is automatically part of the HTTP request of each XML message sent from a client to the Web Services, so this approach imposes no additional burden on the client.

The author implemented this IP address based authentication as a software aspect, utilizing the AspectJ compiler [5] in Java. Authentication is a system level concern, that is, it satisfies a computing goal instead

of a goal of the problem domain that the software is trying to solve--in this case, administering handles. Therefore, IP address based authentication is implemented separately, coded as an aspect that is automatically applied at compile time to the three Web Services that can result in a state change within the local CUL handle server, without directly injecting that authentication code into the source code for the Web Services themselves. Likewise, the functionality to log exceptions or errors that can arise from the use of these Web Services is also implemented as an aspect. Exceptions are sent back to the client in the SOAP response so the client software can decide the appropriate action to take.

Figure 1 provides a high level overview of the handle Web Services design that exposes these simple, platform independent interfaces to the local CUL handle server.

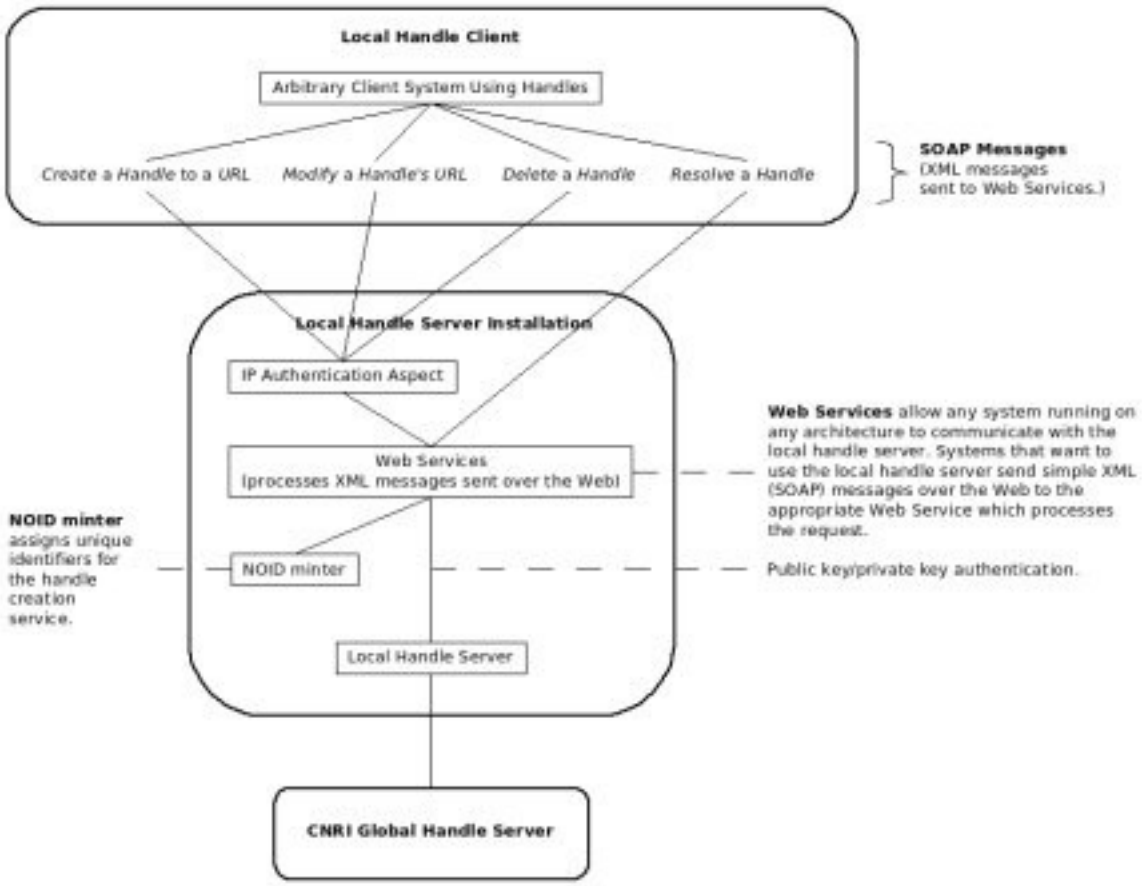

Figure 1.

The server specific details of both the handle server installation and the NOID minter are kept in a configuration file that is loaded by the base Web Service class used for each service. Therefore, by appropriately modifying six entries in this configuration file, these Web Services can easily be redeployed on another CUL server, or at another institution that uses handles.

Finally, the author began creating client libraries for these Web Services that can be used to easily administer handles in other programming environments, without requiring the systems that use these clients to know about the Web Services or to even know about the handle server itself. Currently, Java and Perl clients exist, Python clients are begin developed, and PHP and Ruby clients are planned.

When the Apache Axis toolkit created the Web Services infrastructure for the four interfaces discussed above, it also created corresponding Web Services Description Language (WSDL) files that are published alongside the Web Services so that client software can programmatically discover how to interact with them. Most modern programming languages contain SOAP libraries that can digest these WSDL descriptions programmatically and create local stub methods for the corresponding Web Services that hide

the details of SOAP messaging. The author used such libraries in Java and Perl to create stub methods that were then further simplified so that, for example, a digital library implemented in Java can create a handle for a resource using only one line of code:

```
String assignedPID = HandleClient.create("http://www.library.cornell.edu");
```

As a result, the background network communication with the Web Services and the subsequent communication with the local CUL handle server is transparent to clients using these libraries. The author is committed to creating these client libraries to encourage the adoption of PIDs within CUL.

Implications for Future Development

The author is most immediately exploring the consequences of implementing the IP address based authentication aspect to determine whether a finer grained authentication mechanism is desirable. By trusting traffic from individual IP addresses, multiple clients from any given IP address are effectively indistinguishable, and all clients from valid IP addresses have the same rights to the handle Web Services so that, for example, a client could theoretically modify or even delete a handle that was created by another client on the same machine. So, while IP address authentication removes a significant burden of complexity from the clients, it may also add a burden of institutional responsibility.

Some of these security concerns can be addressed by implementing say, a separate, secure password strategy. Most obviously, the Web Services could then enforce policies such that, for example, only the creator of a particular handle can modify or delete that handle. Such an authentication strategy and enforcement of policies will add complexity to both the implementation of the Web Services and the client code that accesses them, however, and if this strategy were implemented, the author would endeavor to continue hiding as much of this complexity as possible behind friendlier interfaces. Additionally, policies to address the issues of scale inherent in managing client accounts would also be necessary.

Other obvious improvements suggest themselves, such as further generalizing implementation details, and the author has already designed the services with such generalization in mind. For example, a unique identifier interface could be created to abstract this functionality away from the current NOID minter approach so that future implementations could generate identifiers from say, a MySQL database table, without affecting the Web Services themselves that use the generalized interface. Furthermore, this unique identifier interface would likely become its own Web Service independent of those used to administer handles, which would allow multiple instances of those handle Web Services to be deployed in organizations with multiple local handle server implementations, while still ensuring the creation of institutionally unique identifiers through a discrete, localized mechanism.

Because of the Web Services based approach, a similar generalization strategy can be employed for the Handle System itself, so that if in the future, CUL begins using a new PID approach--or even if a new backward incompatible version of the Handle System is deployed--that new PID approach can be implemented behind the same Web Services and transparently become the new preferred PID implementation. Clients of the Web Services would not be required to make any adjustment to such changes, and in fact, the author believes that multiple PID implementations could co-exist simultaneously and transparently behind these Web Services. For example, a client can still administer existing handles while transparently administering newly implemented ARKs, or legacy PURLs using the same Web Services. The client interfaces would be both backward and forward compatible because those clients have no knowledge of the PID implementation underlying the Web Services. Obviously, an assumption underlying this strategy is that the clients determine to what type of resources PIDs should be assigned.

Finally, the author has considered that there may be metadata about the resources for which PIDs are assigned that would be both natural and convenient to administer along with the PIDs themselves. For example, it might be helpful, particularly in a preservation context, to not only resolve a PID to a resource's URI, but also to receive the anticipated checksum for that resource, if applicable, so that after the resource is retrieved, a basic check of its integrity can be performed.

Other use cases might be imagined for other types of metadata. Associating such metadata with PIDs is really not so different than what PIDs are meant to do in the first place. The value of a given PID is that to the client, it is a reliable method of indirection to a resource's current URI. But that URI is only an attribute of that resource, and that attribute is both arbitrary and ephemeral, which is after all why we prefer PIDs.

If the PID strategy stores and retrieves URI metadata for the purpose of accessing the resource, then it may be reasonable to store and retrieve other metadata about the resource alongside its URI to further aid this process, especially when, in the case of a checksum, that metadata is much more closely representative of the resource than is a URI.

A closely related idea is to expand the semantics of what a PID is ultimately referencing. For example, ARKs allow one to optionally resolve a PID to its corresponding digital object data, to the digital object's metadata, or to a commitment statement concerning the digital object, and what the author envisions may look similar to this approach while remaining independent of specific PID implementations. Or, one might simply want to store multiple URI's with a PID to locate different copies of the same resource, such as an access copy and an archival copy of an image, for example.

The Handle System itself is designed with the ability to store arbitrary data in custom fields in a handle's record on the server, and while this ability could accommodate these imagined use cases, storing metadata in this way creates an obvious tight coupling between that metadata and the Handle System. Put another way, in one of the above examples, the goal is to associate a checksum with a PID, not with a handle, which is just a particular PID application. Instead, this metadata, keyed to whatever PID has been assigned, can be stored independently of any particular PID implementation and can be accessed transparently behind the PID Web Services, even if multiple PID implementations are deployed simultaneously. For example, this metadata could be expressed as Resource Description Framework (RDF) triples that are stored and queried using existing RDF software that is integrated into the PID Web Services.

The need for policies and best practices for the form and content of this metadata as well as its maintenance will quickly present itself, but if these challenges can be addressed, it is the author's belief that this collection of Web Services, that easily allows present and future architectures to reliably access resources and corresponding metadata, could form the basis of new and exciting library services based on the resource or digital object as the level of granularity instead of the current collection level view of assets often employed at institutions like CUL. This paradigm shift could not only change how we build new library services, but it could add value to existing collection based systems as well.

The Larger Handle Web Services Discussion

The author is not the first to envision and implement a Web Services interface to the Handle System. For example, at the Digital Library Federation 2006 Fall Forum, Joseph Pawletko from New York University spoke of an XML-RPC based approach to communicating with a handle server in his talk entitled "A DSpace-based Preservation Repository Design."

On the Corporation for National Research Initiatives' (CNRI) "Handle Info" electronic mailing list, several members have either proposed or implemented Web Services for their handle servers. The Australian Persistent Identifiers Linking Infrastructure (PILIN) project [6] is one such example:

> PILIN is piloting a service-oriented persistent identifier infrastructure built on top of Handles. As part of the project we plan to develop web service interfaces to Handle servers, similar to those already discussed on this list. We also plan to develop "value-added" web services that use identifiers to manage Internet resources. (E-mail message from Nigel Ward to handle-info@cnri.reston.va.us on November 29, 2006.)

CNRI itself has also begun work on their own Web Services for administering handles (email message from Giridhar Manepalli to handle-info@cnri.reston.va.us on November 16, 2006) and Linkstorm Corporation has begun work on a REST based approach to do the same (email message from Mark Donoghue to handle-info@cnri.reston.va.us on November 16, 2006). Others on the Handle Info mailing list have proposed creating a working group of those who have implemented, or would like to implement, Web Services for the Handle System to discuss existing implementations, goals and best practices.

Despite the level of interest in Handle System Web Services in general, and the existing private implementations at other institutions, the author developed a core set of Web Services and corresponding client libraries to meet the immediate needs of several ongoing CUL projects and to facilitate the adoption of the Handle System within CUL. As a result, several CUL projects are actively using the Handle System through these Web Services, including CUL's Large Scale Digitization Initiative with Microsoft Corporation.

The lessons learned from the development of these handle services at CUL have been, and will continue to be, added to the larger Handle System Web Services discussion. To that end, the author is currently obtaining permission from Cornell University to freely distribute all of the code discussed in this article, which is possible because the HCL is licensed under a custom agreement similar to the open source BSD license and allows for the creation and subsequent distribution of derivative works based on the HCL source. (Corporation for National Research Initiatives, "HANDLE.NET® SOFTWARE CLIENT LIBRARY (ver. 6) -- Java Version", http://hdl.handle.net/4263537/5024, accessed May 18,2007.) Once the permission to distribute this software is granted by Cornell, a Web site will be launched in Summer 2007 to make this code available and provide community based support and documentation. The author will announce the availability of that site on the Handle-Info mailing list.

Notes

1. HANDLE.NET Software Client Libraries (http://www.handle.net/client_download.html).
2. Noid (Nice Opaque Identifier) Minting and Binding Tool (http://www.cdlib.org/inside/diglib/ark/noid.pdf)
3. Apache Axis (http://ws.apache.org/axis/).
4. Jakarta Tomcat (http://tomcat.apache.org/).
5. AspectJ (http://www.eclipse.org/aspectj/).
6. The Australian Persistent Identifiers Linking Infrastructure (PILIN) project (http://www.arrow.edu.au/PILIN.php).